REMARKS/ARGUMENTS

Receipt of the final Office action dated May 19, 2004 is hereby acknowledged. In that action, the Examiner: 1) rejection claims 1, 2 and 7 as allegedly obvious over Boehm (U.S. Pat. No. 6,457,170) in view of Evans (U.S. Pat. No. 5,805,899) and in further view of Tanaka (U.S. Pat. No. 6,665,735); 2) rejected claim 3 as allegedly obvious over Boehm, Evans, Tanaka and in further view of the Shute article (titled, "Separate Compilation and the UNIX Make Program); 3) rejected claims 4-6 as allegedly obvious over Boehm, Evans, Tanaka, Shute and in further view of the Schach article (titled, "Software Engineering"); 4) rejected claims 9 and 10 as allegedly obvious over Boehm, Evans, and Tanaka in further view of Hunt (U.S. Pat. No. 6,499,137); 5) rejected claim 8 as allegedly obvious over Boehm, Evans and Tanaka in further view of Leblang (U.S. Patent No. 5,649,200); 6) rejected claims 11-13 as allegedly obvious over Boehm, Evans, Tanaka and Hunt in further view Schach; 7) rejected claims 14-17 as allegedly obvious over Evans and Tanaka; 8) rejected claims 14-17 as allegedly lacking utility; and 9) made the action final.

With this Response, Applicant proposes amendments to claims 14-17. Reconsideration is respectfully requested.

I. CLAIM REJECTIONS

A. Claim 1

Claim 1 stands rejected as allegedly obvious over Boehm in view of Evans and in further view of Tanaka.

Boehm appears to disclose a system where version information comes in the form of file names of the various files, as well as some character strings embedded in object files. (*See* Boehm, Col. 4, lines 28-38). The Office action dated May 19, 2004 asserts that Boehm teaches that the version information in the file names finds its way to the executable program, but the location cited (Beohm Col. 8, lines 35-40) does not support this position as it points only to use of the MAKE command. As the Examiner is no doubt aware, the MAKE command is merely a utility to ease creating executable programs from a plurality of source locations. Applicant includes herewith for the Examiner's perusal a description of the MAKE utility available on the Internet.

Evans appears to teach a "mapfile" containing version information.

> In the described embodiment of the invention, **the global symbols and version names associated with each version are defined in a "mapfile" generated by a human being. The mapfile is input to the link-editor at build time along with one, or more, relocatable (compiled) objects to create a versioned object.** By default, at build time, when an application is link-edited with a versioned object that contains versioning information, a dependency is established in the application to those versions that include the global symbols referenced by the application.

Evans, Col. 2, lines 25-34 (emphasis added).

Tanaka appears to be directed to adding preprocessing and/or postprocessing to dynamic link libraries (DLLs) without changing the original DLL file.

> Specifically, a **preprocess or a postprocess can be added to a dynamic link library function and executed without changing the dynamic link library function**, by replacing the dynamic link library name described in the header portion of the program module and[.] a function name, which is to be referred to by the program module, with other names, linking and executing a dynamic link library, which includes a function whose name is the same as the replaced function name and has the same name as the replaced library name, instead of the original dynamic link library, executing the original dynamic link library function using the function having the same name as the replaced function name, and executing the preprocess or postprocess.

(Tanaka Col. 6, lines 20-32 (emphasis added)). Thus, Tanaka's concern is adding pre- and post-processing, not getting version information to the final executable file.

Claim 1, by contrast, recites, "creating a version source file, **the version source file containing a version function whose name comprises at least one of version information and product information of the library**... ." Neither Boehm, Evans nor Tanaka, alone or in combination, teach or fairly suggest that version information should be contained in the name of a function. Claim 1 further recites, "building the executable program to include the library **such that the version function whose name comprises at least one of version information and product information** of the library **is combined into the executable program.**" Even if it is assumed that the version information of Boehm and/or Evans finds its way into the executable program (which Applicant does not admit), the references (including Tanaka) still fail to teach or fairly suggest that version information should be in the form of a function name. Moreover, since Tanaka does not appear to be concerned with version information at all, it is doubtful that Tanaka is even properly combinable with Boehm and/or Evans.

Based on the foregoing, Applicant respectfully submits that claim 1, and all claims which depend from claim 1 (claims 2-13), should be allowed.

B. Claim 14

Claim 14 stands rejected as alleged obvious over Evans in view of Tanaka. Applicant amends claim 14 to address the Section 101 rejection of the Office action dated May 19, 2004.

Evans appears to teach a "mapfile" containing version information. (Evans, Col. 2, lines 25-34). Tanaka appears to be directed to adding preprocessing and/or postprocessing to dynamic link libraries (DLLs) without changing the original DLL file. (Tanaka Col. 6, lines 20-32).

Claim 14, by contrast, specifically recites, "creating source code file **containing a function whose name comprises version information** of a library; compiling the source code file to create an object file placed within the library; and **building an executable program using at least the function from the library such that the version information is contained in the executable program through the presence of the function.**" Even if Evans and Tanaka are properly combined (which Applicant does not admit), the combination does not teach or fairly suggest that the mapfile of Evans could or should be in any way related to the dummy function used to add pre- and post-processing.

Based on the foregoing, Applicant respectfully submits that claim 14, and all claims which depend from claim 14 (claims 15-17), should be allowed.

Appl. No.: 09/764,526
Amdt. dated July 2, 2004
Reply to Office action of May 19, 2004

Claims 15-17 are amended to address the Section 101 rejections, and not to define over any prior art.

II. CONCLUSION

Applicant respectfully requests reconsideration and allowance of the pending claims. If the Examiner feels that a telephone conference would expedite the resolution of this case, he is respectfully requested to contact the undersigned.

In the course of the foregoing discussions, Applicant may have at times referred to claim limitations in shorthand fashion, or may have focused on a particular claim element. This discussion should not be interpreted to mean that the other limitations can be ignored or dismissed. The claims must be viewed as a whole, and each limitation of the claims must be considered when determining the patentability of the claims. Moreover, it should be understood that there may be other distinctions between the claims and the prior art which have yet to be raised, but which may be raised in the future.

If any fees or time extensions are inadvertently omitted or if any fees have been overpaid, please appropriately charge or credit those fees to Hewlett-Packard Company Deposit Account Number 08-2025 and enter any time extension(s) necessary to prevent this case from being abandoned.

Applicant respectfully requests that a timely Notice of Allowance be issued in this case.

Respectfully submitted,

[illegible]

Mark E. Scott, PTO Reg. No. 43,100
CONLEY ROSE, P.C.
(713) 238-8000 (Phone)
(713) 238-8008 (Fax)
ATTORNEY FOR APPLICANT

HEWLETT-PACKARD COMPANY
Intellectual Property Administration
Legal Dept., M/S 35
P.O. Box 272400
Fort Collins, CO 80527-2400

An Introduction to the UNIX Make Utility

Contents

Introduction

This paper is a short introduction to the UNIX `make` utility. The intended audience is computer science students at Middle Tennessee State University (MTSU) of intermediate ability level, if you're taking CSCI 217 this paper will be of use to you. Although `make` can be used in conjunction with most programming languages all examples given here use C++ as this is the most common programming language used at MTSU. It is assumed that you have a good understanding of a C++ compiler. As an introduction this paper intends to teach the reader how to use the most common features of `make`. A more comprehensive guide may be found by examining the list of references provided.

Layout guide

Throughout the paper various text styles will be used to add meaning and focus on key points. All references to the `make` utility, file names and any sample output use the fixed font style, i.e. `fixed font example`. If the example is prefixed with a percent character (`%`) it is a UNIX C-shell command line. Words that are key to `make` terminology are highlighted in **bold** when they occur for the first time.

Overview

The make utility is a software engineering tool for managing and maintaining computer programs. Make provides most help when the program consists of many component files. As the number of files in the program increases so to does the compile time, complexity of compilation command and the likelihood of human error when entering command lines, i.e. typos and missing file names.

By creating a **descriptor file** containing **dependency rules**, **macros** and **suffix rules**, you can instruct make to automatically rebuild your program whenever one of the program's component files is modified. Make is smart enough to only recompile the files that were affected by changes thus saving compile time.

What make does

Make goes through a descriptor file starting with the **target** it is going to create. Make looks at each of the target's **dependencies** to see if they are also listed as targets. It follows the chain of dependencies until it reaches the end of the chain and then begins backing out executing the commands found in each target's rule. Actually every file in the chain may not need to be compiled. Make looks at the time stamp for each file in the chain and compiles from the point that is required to bring every file in the chain up to date. If any file is missing it is updated if possible.

Make builds object files from the source files and then links the object files to create the executable. If a source file is changed only its object file needs to be compiled and then linked into the executable instead of recompiling all the source files.

Simple Example

This is an example descriptor file to build an executable file called `prog1`. It requires the source files `file1.cc`, `file2.cc`, and `file3.cc`. An include file, `mydefs.h`, is required by files `file1.cc` and `file2.cc`. If you wanted to compile this file from the command line using C++ the command would be

```
% CC -o prog1 file1.cc file2.cc file3.cc
```

This command line is rather long to be entered many times as a program is developed and is prone to typing errors. A descriptor file could run the same command better by using the simple command

```
% make prog1
```

or if `prog1` is the first target defined in the descriptor file

```
% make
```

This first example descriptor file is much longer than necessary but is useful for describing what is going on.

```
prog1 : file1.o file2.o file3.o
        CC -o prog1 file1.o file2.o file3.o

file1.o : file1.cc mydefs.h
        CC -c file1.cc

file2.o : file2.cc mydefs.h
        CC -c file2.cc

file3.o : file3.cc
        CC -c file3.cc
```

```
clean :
        rm file1.o file2.o file3.o
```

Let's go through the example to see what make does by executing with the command make prog1 and assuming the program has never been compiled.

1. make finds the target prog1 and sees that it depends on the object files file1.o file2.o file3.o
2. make next looks to see if any of the three object files are listed as targets. They are so make looks at each target to see what it depends on. make sees that file1.o depends on the files file1.cc and mydefs.h.
3. Now make looks to see if either of these files are listed as targets and since they aren't it executes the commands given in file1.o's rule and compiles file1.cc to get the object file.
4. make looks at the targets file2.o and file3.o and compiles these object files in a similar fashion.
5. make now has all the object files required to make prog1 and does so by executing the commands in its rule.

You probably noticed we did not use the target, clean, it is called a **phony target** and is explained in the section on dependency rules.

This example can be simplified somewhat by defining macros. Macros are useful for replacing duplicate entries. The object files in this example were used three times, creating a macro can save a little typing. Plus and probably more importantly, if the objects change, the descriptor file can be updated by just changing the object definition.

```
OBJS = file1.o file2.o file3.o

prog1 : $(OBJS)
      CC -o prog1 $(OBJS)

file1.o : file1.cc mydefs.h
      CC -c file1.cc

file2.o : file2.cc mydefs.h
      CC -c file2.cc

file3.o : file3.cc
      CC -c file3.cc

clean :
      rm $(OBJS)
```

This descriptor file is still longer than necessary and can be shortened by letting make use its internal macros, special macros, and suffix rules.

```
OBJS = file1.o file2.o file3.o

prog1 : $(OBJS)
      $(CXX) -o $@ $(OBJS)

file1.o file2.o : mydefs.h

clean :
      rm $(OBJS)
```

Invoking make

Make is invoked from a command line with the following format

```
make [-f makefile] [-bBdeiknpqrsSt] [macro name=value] [names]
```

However from this vast array of possible options only the -f makefile and the names options are used frequently. The table below shows the results of executing make with these options.

Command	Result
make	use the default descriptor file, build the first target in the file
make myprog	use the default descriptor file, build the target myprog
make -f mymakefile	use the file mymakefile as the descriptor file, build the first target in the file
make -f mymakefile myprog	use the file mymakefile as the descriptor file, build the target myprog

Frequently used make options

When using a default descriptor file make will search the current working directory for one of the following files in order:

```
makefile
Makefile
```

Hint: use Makefile so that it will list near the beginning of the directory and be easy to find.

Descriptor files

To operate make needs to know the relationship between your program's component files and the commands to update each file. This information is contained in a descriptor file you must write called Makefile or makefile.

Comments

Comments can be entered in the descriptor file following a pound sign (#) and the remainder of the line will be ignored by make. If multiple lines are needed each line must begin with the pound sign.

```
# This is a comment line
```

Dependency rules

A rule consist of three parts, one or more targets, zero or more dependencies, and zero or more commands in the following form:

```
target1 [target2 ...] :[:] [dependency1 ...] [; commands]
        [<tab> command]
```

Note: each command line must begin with a tab as the first character on the line and only command lines may begin with a tab.

Target

A target is usually the name of the file that make creates, often an object file or executable program.

Phony target

A phony target is one that isn't really the name of a file. It will only have a list of commands and no prerequisites. One common use of phony targets is for removing files that are no longer needed after a program has been made. The following example simply removes all object files found in the directory containing the descriptor file.

```
clean :
   rm *.o
```

Dependencies

A dependency identifies a file that is used to create another file. For example a .cc file is used to create a .o, which is used to create an executable file.

Commands

Each command in a rule is interpreted by a shell to be executed. By default make uses the /bin/sh shell. The default can be over ridden by using the macro SHELL = /bin/sh or equivalent to use the shell of your preference. This macro should be included in every descriptor file to make sure the same shell is used each time the descriptor file is executed.

Macros

Macros allow you to define constants. By using macros you can avoid repeating text entries and make descriptor files easier to modify. Macro definitions have the form

```
NAME1 = text string
NAME2 = another string
```

Macros are referred to by placing the name in either parentheses or curly braces and preceding it with a dollar sign ($). The previous definitions could referenced

```
$(NAME1)
$(NAME2)
```

which are interpreted as

```
text string
another string
```

Some valid macro definitions are

```
LIBS = -lm
OBJS = file1.o file2.o $(more_objs)
more_objs = file3.o
CXX = CC
DEBUG_FLAG =          # assign -g for debugging
```

which could be used in a descriptor file entry like this

```
prog1 : $(objs)
```

```
        ${CXX} $(DEBUG_FLAG) -o progl ${objs} ${LIBS}
```

Macro names can use any combination of upper and lowercase letters, digits and underlines. By convention macro names are in uppercase. The text string can also be null as in the DEBUG_FLAG example which also shows that comments can follow a definition.

You should note from the previous example that the OBJSmacro contains another macro $(MORE_OBJS). The order that the macros are defined in does not matter but if a macro name is defined twice only the last one defined will be used. Macros cannot be undefined and then redefined as something else.

Make can receive macros from four sources, macros maybe defined in the descriptor file like we've already seen, internally defined within make, defined in the command line, or inherited from shell environment variables.

Internal macros

Internally defined macros are ones that are predefined in make. You can invoke make with the -p option to display a listing of all the macros, suffix rules and targets in effect for the current build. Here is a partial listing with the default macros from MTSU's mainframe **frank**.

```
CXX = CC
CXXFLAGS = -O
GFLAGS =
CFLAGS = -O
CC = cc
LDFLAGS =
LD = ld
LFLAGS =
MAKE = make
MAKEFLAGS = b
```

Special macros

There are a few special internal macros that make defines for each dependency line. Most are beyond the scope of this document but one is especially useful in a descriptor file and you are likely to see it even in simple descriptor files.

The macro @ evaluates to the name of the current target. In the following example the target name is progl which is also needed in the command line to name the executable file. In this example -o @ evaluates to -o progl.

```
progl : ${objs}
    ${CXX} -o $@ ${objs}
```

Command line macros

Macros can be defined on the command line. From the previous example the debug flag, which was null, could be set from the command line with the command

```
% make progl DEBUG_FLAG=-g
```

Definitions comprised of several words must be enclosed in single or double quotes so that the shell will pass them as a single argument. For example

```
    % make prog1 "LIBS= -lm -lX11"
```

could be used to link an executable using the math and X Windows libraries.

Shell variables

Shell variables that have been defined as part of the environment are available to make as macros within a descriptor file. C shell users can see the environment variables they have defined from the command line with the command

```
    % env
```

These variables can be set within the .login file or from the command line with a command like:

```
    % setenv DIR /usr/bin
```

Macro assignment priority

With four sources for macros there is always the possibility of conflicts. There are two orders of priority available for make. The default priority order from least to greatest is:

1. internal definitions
2. shell environment variables
3. descriptor file definitions
4. command line macro definitions

If make is invoked with the -e option the priority order from least to greatest is

1. internal definitions
2. descriptor file definitions
3. shell environment variables
4. command line macro definitions

Suffix rules

Make has a set of default rules called suffix or implicit rules. These are generalized rules that make can use to build a program. For example in building a C++ program these rules tell make that .o object files are made from .cc source files. The suffix rule that make uses for a C++ program is

```
    .cc.o:
        $(CXX) $(CXXFLAGS) -c $<
```

where $< is a special macro which in this case stands for a .cc file that is used to produce a particular target .o file.

References

1. Becker, B. (Jan, 1996). A GNU Make Tutorial
   http://www.undergrad.math.uwaterloo.ca/~cs241/make/tutorial/index.html
2. Hewlett-Packard, (Nov, 1993). make (1) man pages. 16 pp.
3. Oram,A. & Talbott, S. (Feb, 1993). Managing Projects with make. 149 pp.

4. Stallman, R & McGrath Roland (Dec, 1993). GNU Make http://csugrad.cs.vt.edu/manuals/make/make_toc.html
5. Wang, P. (1993). ANSI C on UNIX. 432 pp.